NASA Contractor Report 178129

ICASE REPORT NO. 86-46

ICASE

STATISTICAL METHODOLOGIES FOR THE CONTROL
OF DYNAMIC REMAPPING

Joel H. Saltz
David M. Nicol

Contract No. NAS1-18107
July 1986

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

Statistical Methodologies for the Control of Dynamic Remapping

Joel H. Saltz
David M. Nicol

Institute for Computer Applications in Science and Engineering

Abstract

Following an initial mapping of a problem onto a multiprocessor machine or computer network, system performance often deteriorates with time. In order to maintain high performance, it may be necessary to remap the problem. The decision to remap must take into account measurements of performance deterioration, the cost of remapping, and the estimated benefits achieved by remapping. We examine the tradeoff between the costs and the benefits of remapping two qualitatively different kinds of problems. One problem assumes that performance deteriorates gradually, the other assumes that performance deteriorates suddenly. We consider a variety of policies for governing when to remap. In order to evaluate these policies, statistical models of problem behaviors are developed. Simulation results are presented which compare simple policies with computationally expensive optimal decision policies; these results demonstrate that for each problem type, the proposed simple policies are effective and robust.

This research was supported by the National Aeronautics and Space Administration under NASA Contract Number NAS1-18107 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

1. Introduction

The performance of message passing multiprocessors and of distributed systems often hinges on the way in which data is distributed throughout the memories of the individual processors in the system or network. In many computational problems a discrete model of a physical system is assumed, and a set of values is calculated for every domain point in the model. Portions of a domain are assigned to the memory of each processor, and each processor assumes responsibility for calculations that relate to the subdomain assigned to its memory. As the problem proceeds, for reasons to be discussed later, changes may occur in the amount of work required for the calculations and communications pertaining to the portion of the problem assigned to a processor. Due to the coupling between subdomains, the rate at which progress is made in solving the problem is limited by the maximally loaded processor. System performance deteriorates in time, often in a relatively gradual way, and at some point a remapping of the problem onto the processors may be advantageous. We shall call these types of problems *varying demand distribution problems.* Statistical models are developed of the behavior of this kind of problem and through the use of these models, policies are evaluated for deciding when remapping should occur.

In a wide variety of applications, the patterns of processor use and memory access are determined directly or indirectly by a population of users or programs whose requirements are difficult to anticipate. Database management systems are examples of such applications. When a database management system is mapped onto computer systems so that responsibility for computations and communications pertaining to subsets of the stored information is assigned to a given processor, the distribution of work among the processors is highly dependent on the nature of the queries received. A related problem occurs in the parallel simulation of digital circuitry. When responsibility for computations concerning particular circuit elements are assigned to each processor, the distribution of work among processors becomes highly dependent on the portions of the circuit currently undergoing testing. The system per-

formance is closely linked to the distribution of work among the processors, and this distribution depends on often unpredictable inputs to the program.

Once a partition is chosen, it is necessary to monitor the performance of the system to determine whether changes in the nature of the inputs have caused a statistically demonstrable deterioration in system performance. The repartitioning of these applications may cost substantial overhead and consequently should be undertaken only when an overall benefit can be anticipated. If deterioration in system performance has taken place, it needs to be determined whether repartitioning might be worthwhile. A period of time in which a program exhibits no statistically demonstrable change in behavior will be called a *phase*. Problems which consist of a number of consecutive phases will be called *multiphase* problems. In this paper we shall discuss probabilistic models that describe varying demand distribution problems and problems exhibiting multiphase behavior. Policies for weighing the costs and benefits of partitioning for both types of problems will be evaluated.

2. Varying Demand Distribution Problems

Varying demand distribution problems assume a discrete model of a physical system, and calculate a set of values for every domain point in the model. These values are often functions of time, so that it is intuitive to think of the computation as marching through time. When such a problem is mapped onto a message passing multiprocessor machine, or a shared memory machine with fast local memories, regions of the model domain are assigned to each processor. The running behavior of such a system is often characterized as a sequence of steps, or iterations. During a step, a processor computes the appropriate values for its domain points. At the step's end, it communicates any newly computed results required by other processors. Finally, it waits for other processors to complete their computation step, and send it data required for the next step's computation.

The computational work associated with each portion of a problem's subdomain may change over the course of solving the problem. This may be true if the behavior of the modeled physical system

changes with time. The distribution of computational work over a domain may also change in problems without explicit time dependence because during the course of solving a problem, for example, more work may be required to resolve features of the emerging solution. Since time stepping is often used as a means for obtaining a steady state solution, there is considerable overlap between the above mentioned categories. Because of the synchronization between steps, the system execution time during a step is effectively determined by the execution time of the slowest, or most heavily loaded processor. We can then expect system performance to deteriorate in time, as the changing resource demand causes some processor to become proportionally overloaded. One way of dealing with this problem is to periodically redistribute, or remap load among processors.

Changing distributions of computational work over a domain arise through the use of adaptive methods in the solution of hyperbolic partial differential equations in which extra grid points are placed in some regions of the problem domain in order to resolve all features of the solution to the same accuracy [1],[2],[3],[4], [5],[6]. A number of studies have investigated methods for redistributing load in message passing processors for this type of problem [7],[8],[9]. Changing distributions of computational work can also occur when vortex methods are applied to the numerical simulation of incompressible flow fields. In these methods, invicid fluid dynamics is modeled by parcels of vorticity which induce motion in one another [10], [11]. The number of vortices corresponding to a given region in the domain varies during the course of the solution of a problem. Methods for dynamically redistributing work in this problem have been investigated [12].

In multirate methods for the solution of systems of ordinary differential equations[13], different variables in the system of equations are stepped forward with different timesteps. The size of the timesteps in the system is generally equal to that of a globally defined largest timestep divided by an integer. The size of the different timesteps utilized may vary during the course of solving the problem, and hence the computational work associated with the integration of a given set of variables may

change. Methods for solving elliptic partial differential equations have been developed where iterations on a sequence of adaptively defined meshes are carried out [14],[15],[16],[17],[18] Generally both the total amount of computational work required by each of the meshes and the distribution of work within the domain changes as one moves from one mesh to the next. In time driven discrete event simulations [19], one simulates the interactions over time of a set of objects. Responsibility for a subset of objects is assigned to each processor. Over the course of the simulation, subsets of objects may differ in activity, and hence in their computational requirements. This problem may also arise in parallel simulations which proceed in a loosely synchronized manner, such as those described in [20], [21], [22], [23], [24], [25].

There are two fundamentally different approaches to such remapping. The decentralized load balancing approach is usually studied in the context of a queueing network [26],[27],[28],[29],[30],[31], [32]. Load balancing questions are then focused on "transfer policies", and "location policies"[26]. A transfer policy governs whether a job arriving at a service center is kept or is routed elsewhere for processing. A location policy determines which service center receives a transferred job. Decentralized balancing seems to be the natural approach when jobs are independent, and when a global view of balancing would not yield substantially better load distributions.

However, a large class of computations is not well characterized by a job arrival model, and it may be advantageous to take a global, or centralized perspective when balancing. We will call a global balancing mechanism "mapping" to distinguish it from the localized connotations of the term load balancing. A centralized mapping mechanism can exploit full knowledge of the computation and its behavior. Furthermore, dependencies between different parts of a computation can be complex, making it difficult to dynamically move small pieces of the computation from processor to processor in a decentralized way. Global mapping is natural in a computational environment where other decisions are already made globally, e.g. convergence checking in an iterative numerical method. Yet the execu-

tion of a global mapping algorithm may be costly, as may the subsequent implementation of the new workload distribution. A number of authors have considered global mapping policies under varying model assumptions, for example, see [33], [34], [35], [36], [37], [38], and a comparison between global and decentralized mapping strategies is reported in [39].

For the types of problems we describe, *remapping* the load with a global mechanism is tantamount to repartitioning the set of model domain points in regions, and assigning the newly defined regions to processors. A mapping algorithm of this sort is studied in [7] and the performance of this mapping algorithm in the context of vortex methods is investigated in [12]. Decision policies determining *when* a load should be remapped become quite important. The overhead associated with remapping can be high, so it is important to balance the overhead cost of remapping with the expected performance gain achieved by remapping. While this is a generic problem, the details of load evolution, of the remapping mechanism, and of various overhead costs are system and computation dependent. In order to study general properties of remapping decision policies, it is necessary to *model* the behavior of interest, and evaluate the performance of decision policies on those models. In the present paper we consider remapping of varying demand distribution problems using three different stochastic models.

The evaluation of policies for memory management in multiprogrammed uniprocessor systems has successfully employed a number of stochastic models to reflect the memory requirements of typical programs [40],[41]. In these models, the principal of memory reference locality plays a central role. Evaluating policies for scheduling a remapping in message passing machines is somewhat similar in spirit to the evaluation of paging algorithms in multiprogrammed uniprocessor systems. The principal of locality that we attempt to capture here is the locality of resource demand. The computational work corresponding to the problem region assigned to a given processor will often vary in a gradual fashion. In this paper we consider two models which describe this evolution probabilistically. The first model assumes that the computational requirements of each partition region behaves as a Markov chain,

independently of any other region; this is called the Multiple Markov chain (MUM) model. The MUM model has the advantage of being analytically tractable in several ways. However, for many problems it may not be reasonable to assume independence in load evolution between partition regions. We address this issue with a second, less tractable model, the Load Dependency (LD) model. These models attempt to capture the dynamics by which the distribution of computational load changes in time, and are characterized by a small number of important parameters. Through the use of these load evolution models, we are able to evaluate policies for deciding when load should be remapped.

3. The Multiple Markov Chain Load Model

We describe both the drifting load and synchronization aspects of varying demand distribution problems with the Multiple Markov chain (MUM) model. MUM characterizes a processor's changing load by a discrete Markovian birth-death process. The state s of the chain is a positive integer describing the execution time of the processor at a step. We also assume that $s \leq L$ for some L. The transition probabilities out of s reflect the principle of locality, where all one step transitions are to neighboring states. When s is between 2 and L–1, the probability that the chain will make a one step transition to state s+1 is $p/2$, the probability of a one step transition to s–1 is $p/2$ and the probability that the chain will not make a transition is 1–p. For $s = 1$ or $s = L$, the state remains the same with probability $1 - p/2$, and moves to the single neighboring state with probability $p/2$.

A system of N parallel processors is modeled by an independent and identically distributed collection of birth-death processes. We let $T_j(n)$ represent the time required by the *jth* processor to complete the *nth* step. The time required for the system as a whole to complete the *nth* step is given by

$$T_{\max}(n) = \max_{1 \leq j \leq N} \{T_j(n)\}.$$

We thus model synchronization by requiring the system to wait for the longest running processor. The average processor execution time during the *nth* step is simply

$$\overline{T}(n) = \frac{1}{N}\sum_{j=1}^{N} T_j(n).$$

One measure of system performance during the *nth* step is the difference between the system execution time and average processor execution time at that step. In fact, this difference plays a key role in our remapping decision policy. Another parameter playing a key role is the cost of remapping once, denoted by C. This cost includes both the communication costs and the computational overhead required for performing a remapping operation. Note that for the sake of tractability, the cost of remapping is assumed here to be independent of the way in which load is distributed at the time remapping occurs. The cost of remapping may be affected in a problem and machine dependent way by the distribution of load prior to balancing.

Figure 1a depicts the behavior of the MUM model for varying numbers of chains. The performance shown is the average (per step) processor utilization as a function of step, taken over 500 simulations or sample paths, where $p = 0.5$ and each chain has 19 states. Performance declines more quickly and to a lower level as one simulates a problem with an increasingly large number of independent processors. For a given number of chains, the performance decline arises from the fact that the expected value of $\overline{T}(n)$ remains relatively constant as n increases, while the expected value of $T_{\max}(n)$ increases in n. Figure 1b depicts the performance of single sample paths of the MUM model using varying numbers of chains, where as before $p = 0.5$ and each chain has 19 states. Note that the decline in performance as a function of step is true only in the sense of comprising a long term trend; each curve has many local maxima and minima. This point is particularly important, because any dynamic real time remapping policy mechanism is concerned with the current single sample path defined by the computation's execution.

In considering the performance of the MUM model, both C and the difference $T_{\max}(n) - \overline{T}(n)$ are viewed as costs; if the computation is first remapped at step n, then the average cost per step is denoted $W(n)$, and is given by

$$W(n) = \frac{\sum_{j=1}^{n}(T_{max}(j) - \overline{T}(j)) + C}{n}.$$

In [42] we show that that for the MUM model, as long as the Markov chains are not extremely active, $E[T_{max}(n)]$ - $E[\overline{T}(n)]$ is an increasing function of n. It is also shown there that if $E[T_{max}(n)]$ - $E[\overline{T}(n)]$ is an increasing function of n, then $E[W(n)]$ has at most one local minimum. Furthermore, if that minimum exists at $\hat{n}$, and if remapping "resets" the behavior of $E[T_{max}(n)]$ - $E[\overline{T}(n)]$, then the optimal fixed-interval remapping policy (which includes never remapping) is to remap every $\hat{n}$ steps.

An intuitively appealing remapping decision policy is to monitor system performance and to remap when the experimentally measured $W(n)$ is minimized. Even though the expected value of $W(n)$ may have at most one minimum, we have no such guarantee for a given simulation or sample path. The Stop at Rise (SAR) remapping policy chooses to remap when the first local minimum m of $W(n)$ is detected. Note that we cannot know that a local minimum of $W(n)$ was achieved at step m until step m+1, consequently we must remap at m+1 and achieve average cost $W(m+1)$.

Figure 2 shows the values of $W(n)$ achieved by simulation runs based on the MUM model when $p = 0.5$, $L = 19$, and $C = 8$. It is interesting to note that the value of $W(n)$ at its first local minimum tends to be quite close to the minimum value of $W(n)$. For a variety of MUM model parameter values, simulations were carried out comparing the expected value of the global minimum of $W(n)$ to the expected value of the local minimum of $W(n)$. The differences between the global and the local minima were observed to quite minor.

We studied the performance of the SAR policy by comparing it to three other policies: the optimal policy, the "remap every m steps" policy, and the policy which never remaps. It is possible to compute the expected time required to complete small sized problems when the optimal Markov decision policy is utilized to decide when to remap. Figure 3 compares a performance metric for the Markov decision policy, SAR, and a non-remapped system for three chains, 100 steps, and various

remapping costs. The SAR data is the average of 500 simulation runs for each value of C. As the remapping cost increases, the discrepancy between the performance obtained through the optimal decision policy and SAR increases. With increasing remapping cost, both the performance of the optimal decision policy and of SAR approach the performance obtained when no remapping is performed.

The performance metric used in figure 3 for all policies depicted in that figure is an estimate of processor utilization: the *ratio* of the *expected* $\sum_{j=1}^{n}\overline{T}(j)$ to the *expected* total time spent by the system to solve the problem, including the cost of all remappings. This measure is useful in figure 3 as it is straightforward in the case of the optimal policy to calculate the expected time required to complete a problem as well as the expected $\sum_{j=1}^{n}\overline{T}(j)$. For all subsequent figures, performance data is obtained by simulation, and the easily computed average performance over all simulations is utilized, i.e. the *mean* of the *ratio* of the $\sum_{j=1}^{n}\overline{T}(j)$ to the total time spent by the system to solve the problem, including the cost of all remappings. Both performance measures were computed for all simulations, and found to differ from each other by less than one percent.

One simple but intuitive remapping policy is the "remap every m steps" policy, or *fixed interval* policy. This policy is insensitive to statistical variations in a system's performance, and requires pre-run-time analysis to determine an effective value of m. However, we might well choose to employ a fixed interval policy if it is costly to measure system performance at every step. In this case, we would attempt to choose m to optimize the system's *expected* performance. In figure 4, we compare the performance obtained through the use of: (1) SAR , (2) the fixed interval policy for a wide range of values of m, and (3) not remapping at all. The performance obtained in a system using the MUM model with eight independent processors is depicted. Each problem consists of 400 steps, each data point is obtained through 200 simulations, and remapping costs of 2 and 8 are assumed. For the SAR policy, we plotted performance against the calculated average number of steps between consecutive

remappings. In the fixed interval policy, we plotted performance against m, the fixed number of steps between remappings. The number of steps between remappings has no meaning when no remapping is done, the performance obtained when no remapping occurs is plotted as a straight horizontal line to facilitate comparison with the other results. The calculation of the performance obtained through the use of the optimal Markov decision policy is not practical in this case due to the long run times and large memory requirements that would be required.

It is notable that SAR's performance was comparable and in fact slightly higher than that obtained by remapping at the optimal fixed interval. The average number of elapsed steps between SAR remappings corresponds closely to the optimal fixed interval remapping policy. Similar results were obtained in other cases using the MUM load model. These results are encouraging for two reasons. Since SAR adapts to statistical variations in the system's behavior, we would hope that it can outperform a non-adaptive policy. Our data shows that SAR outperforms the **optimal** fixed interval policy. Secondly, SAR appears to find the "natural frequency" of remapping for a given remapping cost. While the exact number of steps between remappings may vary with the system's sample path, the average number of steps between remappings is close to that of the optimal fixed interval policy. Note also that the performance obtained by SAR is markedly superior to the performance obtained when no remapping is performed. From extensive simulation results not presented here, we found that the difference between the performance obtained by SAR and the performance obtained when no remapping is performed increases with the number of chains. This is consistent with the observed results in figures 3 and 4.

In the face of uncertainity about future problem behavior, it is reasonable to design a remapping decision policy which optimizes performance locally in time. The SAR policy does this by attempting to minimize $W(n)$, a statistic which measures performance since the last remapping. Performance experiments show that the SAR policy effectively finds the "natural frequency" of remapping as a

function of the rate at which resource demand changes, and the cost of remapping. As such, SAR is a promising policy for real remapping situations. We will next demonstrate that the SAR policy can also be effectively employed with computational models other than the MUM model.

4. Load Dependency Model

While the MUM model is analytically tractable, some of its assumptions may not be realized in practice. For example, MUM assumes that a processor's load drift is stochastically independent of any other processor's load drift. It is easy to construct examples where this assumption is violated. This flaw could be corrected by allowing correlation between chains' transitions, but then an appropriate model of correlation would have to be determined. MUM also assumes homogeneous Markov chains; there is no problem in allowing heterogeneous chains, but the analysis we have developed does not apply to such a model. More seriously, the MUM model implicitly assumes that the transitional behavior of a processor's computational load is determined by the processor, rather than the load. This flaw is corrected in a model where the distinction between a processor and its load is clearly drawn. We call this the Load Dependency (LD) model.

The LD model directly simulates the spatial distribution of computational load in a domain. We consider a two dimensional plane in which activity occurs, for example, a factory floor. To simulate this activity we impose a dense regular grid upon the plane; each square of the grid defines an *activity point.* We suppose that activity in the plane is discretized in simulation time, and model the behavior of activity as follows. Each time step a certain amount of activity may occur at an activity point. This activity is simulated (for example, arrival of parts to a manufacturing assembly station), causing a certain amount of computation. By the next time step some of that activity may have moved to neighboring activity points. This movement of activity simulates the movement of physical objects in a physical domain, and is modeled by the movement of *work units.* A work unit is always positioned at some activity point, and has a weight describing its computational demand at that activity point. From one

time step to the next, a work unit may move from an activity point to a neighboring activity point; this movement is governed probabilistically. In the LD model, the probability that a work unit will move from one activity point to another, as the problem goes from one time step to the next is called the *transition probability* linking the two activity points.

We employ binary dissection [7] to partition the activity points into *N activity regions*, where the points in an activity region form a rectangular mass. The weight of an activity point is taken to be the sum of the weights of work units at the point, at the time that the partitioning is performed. The computational load on a processor during a time step is found by adding the weights of all work units resident on activity points assigned to that processor.

In a wide variety of problems, including those mentioned in section 1 as examples of varying demand distribution problems, data dependencies are quite local. Decomposition of a domain into contiguous regions with a relatively small perimeter to area ratio is thus generally desirable for reducing the quantity of information that must be exchanged between partitions. Furthermore, due to the local nature of the data dependencies, the communication required between partitions in a binary dissection will generally be greatest in partitions that are in physical proximity. The analysis in [7] shows that this type of partition is effective for static remapping, and is easily mapped onto various types of parallel architectures. Estimates are also obtained of the communication costs incurred when binary dissection is used to partition a problem's domain, and the resulting partitions are mapped onto a given architecture. The communication cost estimates obtained by such analysis are inevitably problem, mapping and architecture dependent.

A processor's load changes from one time step to the next when a work unit either moves to an activity point assigned to another processor, or similarly moves from an activity point in a different processor. This explicit modeling of work unit movement removes the most serious flaw with the MUM model. Unlike the MUM model, the change in a processor's computational load from one time

step to the next is explicitly dependent on its own load, and on the loads of processors with neighboring activity regions.

To ensure the correctness of the simulation, we require that all computation associated with a time step be completed before the simulation advances to the next time step. Thus, as in the MUM model, the time required to complete a time step is the maximum computation time among all processors. Again like the MUM model, as time progresses any initial balance will disappear, and average processor utilization will drop. This is particularly true if the work unit movement probabilities are anisotropic, i.e. work movement probabilities vary with the direction of movement.

The SAR policy can also be used with the LD model, since the $W(n)$ statistic requires only the mean processor execution time, the maximum processor execution time per time step, and the remapping cost C. The performance of SAR on the LD model was examined by once again comparing SAR to the performance of fixed interval polices. Figure 5 plots expected processor utilization as a function of time for remapping costs of 50 and 100 work units, when a 64 by 64 mesh of activity points is initialized with one work unit per activity point, and 16 processors are employed. The transition probabilities are anisotropic (given in the figure legend), so that the work tends to drift to the upper right portion of the mesh over time. Not taken into account here is the cost of the interprocessor communication that occurs at the end of each step when partitions exchange newly computed results. As was observed in the MUM model the the performance of the SAR rule and the average number of elapsed steps between SAR remappings corresponded closely to that of the fixed interval leading to the optimal performance. In figure 5, the performance of SAR for a given cost is superior to that obtained from fixed load balancing at the optimal frequency. In other simulations, the performance obtained from SAR was comparable to, but slightly below that obtained from the optimal fixed load balancing method. Note that the performance obtained by SAR in figure 5 is markedly greater than that obtained when no remapping is performed. We shall now shift attention to the other type of multiprocessor per-

formance pattern we are considering here. In this case we will also describe a probabilistic model for this type of problem and propose and evaluate policies for weighing partitioning costs and benefits.

5. Multiphase Problems

The patterns of computation and memory access of many programs are highly dependent on data that become available only at run-time. The effective partitioning of a database between a number of communicating processes is a function of the information stored in the database as well as the transactions that are to be executed. Methods for partitioning logical objects or relations among processors involve deciding how to partition the relation and how to assign the resulting partitions to processors [43],[44]. A related area of active research has involved finding the optimal distribution of files among processors in a message passing machine or computer network, given a set of transactions involving the files. For a review of this work see [45].

A similar problem occurs during the computer aided design and design testing of VLSI components. Testing is generally performed using computer simulation, for which the time costs often grow exponentially with the number of components to be simulated. A logic network can be characterized as a collection of functional units whose executions are constrained by precedence relations defined by input and output lines. Examples of such functional units are AND and OR gates, a more complex functional unit might be an instruction decoder. Typical logic simulators will simulate a functional unit only when at least one of its input values change. Hence in this case as well, the performance achievable by any given problem partitioning is sensitive to the probabilistic distribution of input values [25].

Once a partition is chosen, monitoring the performance of the system to determine whether a phase change has occurred may be essential for the efficient functioning of the multiprocessor system. If a computation mapped onto a parallel computer experiences a phase change, the computation's mapping may no longer effectively exploit parallelism, particularly if the mapping was chosen to

optimize performance during a previous phase. It may then be desirable to remap the computation when a phase change occurs. However, it is generally unreasonable to assume that phase changes can be predicted and identified a priori. It is necessary then to detect and react to phase changes as they occur. However, several problems exist which must be considered. The phase change detection mechanism may not be completely reliable; it may report a phase change when none has occurred, or fail to report a true phase change. Furthermore, the delay cost of remapping the computation may be high; the cost of remapping must be balanced against the resulting expected performance gains. We have treated these problems in the framework of a Markov Decision Process. Within this framework, we are able to demonstrate the structure of a decision policy which minimizes the computation's expected execution time. One of our major results is that the optimal decision policy is a threshold policy: we remap when the probability that a phase change has occurred is high enough. Our second major result is that optimal performance is relatively insensitive to precise estimation of the parameters which determine the optimal policy. Our empirical work suggests that the dominant issue in this problem is the accurate detection of phase changes. We next describe our model of the problem, and present these results.

6. Phase Change Model

Our model of the problem makes several simplifying assumptions. First, we suppose that the computation will undergo at most one phase change. We will later argue that our empirical data shows that further mathematical detail is unnecessary. Note that this assumption differs from the MUM or LD models of load drift we have already discussed. In the current model, a change in behavior is assumed to occur abruptly, rather than gradually. A significant assumption is that the computation's behavior can be described as consisting of a number of *cycles*. For a given problem, a cycle represents a specified unit of work, and at the beginning of each cycle we assume that a measurement of system performance is available. We assume the existence of a phase change detection mechanism. Because

the detection of a phase change is likely to be problem and system dependent, we do not attempt to treat change detection other than to assume that the mechanism is not completely reliable. We model its unreliability by assuming that every invocation of the detection mechanism has a probability α of reporting that a phase change exists when it actually does not exist, and a probability β of it failing to report an existing phase change.

Our remapping decision process will invoke the phase change detection mechanism at the beginning of every cycle. To model the uncertainity in the occurrence of the phase change, we assume that the probability of a phase change having occurred since the last change test is ϕ. When invoked, the change detection mechanism attempts to determine whether the phase has changed any time in the past, presumably on the basis of the computation's recent behavior. Upon receiving the mechanism's report, the decision process will update the probability of a phase change having already occurred. This probability is a function of the mechanism's report, α, β, ϕ, and a prior probability of change. The decision process then decides whether to remap on the basis of the calculated probability. For this reason, we call the sequence of change test and decision a *decision step*. We will let p_n denote the probability of change which is calculated by the *nth* decision step.

We intuitively understand that the decision to remap should be a function of the costs and benefits of remapping. We next define model parameters which capture these costs and benefits. We make the reasonable assumption that the time required for the system to execute a cycle depends on the mapping, and whether the mapping was designed to exploit parallelism during that cycle's phase. We let e_O denote the mean time to execute a cycle before a phase change. We let e_B denote the mean time to execute a cycle after a phase change, but before a remapping. We let e_R denote the mean time to execute a cycle after a phase change, and after a remapping. We suppose that $e_R < e_B$, which simply says that remapping after a phase change leads to higher performance during the new phase. In fact, the difference $e_B - e_R$ is interpreted as the per cycle cost of not remapping after a phase change. We

also explicitly identify the costs of remapping. We let D_d be the expected time delay of calculating and testing a new mapping. For reasons which will become apparent later, we separate this cost from the cost D_r of actually implementing the new mapping.

We also contend that the decision to remap should consider the remaining length of the computation. If the remaining computation is quite short, then it is likely that the benefits of remapping after a phase change cannot outweigh remapping's overhead costs; conversely, a long computation may admit remapping early in the computation even if the per cycle performance gain is small. We model our uncertainity in the duration of the computation by supposing that the computation requires a random number of cycles, denoted M. For tractability reasons, we assume that M is bounded from above by some constant, and that M has an increasing failure rate function. The latter requirement simply means that the longer the computation runs, it becomes more likely that the computation will terminate with the next cycle. Table I below summarizes our model parameter definitions. After the *nth* cycle, the decision process invokes the phase change detection mechanism and calculates a new probability p_n of the phase change having occurred. Our model does not attempt to capture the cost of invoking the

Table I

Model Parameter Definitions

Notation	Definition
n	Decision Step Number
M	Random number of decision steps
e_O	Decision Interval Pre-Change Execution Time, Original Partition
e_B	Decision Interval Post-Change Execution Time, Original Partition
e_R	Decision Interval Post-Change Execution Time, New Partition
D_d	Delay to Calculate and Test New Partition
D_r	Delay to Implement New Partition
α	Change Test False Positive Error
β	Change Test False Negative Error
ϕ	Time of Change Failure Rate Probability

mechanism. On the basis of the value of p_n, the decision process decides whether to generate a new mapping (based on the computation's post-change behavior). The decision to retain the current mapping causes no further activity until the end of the next cycle. If the phase change has not occurred, the expected execution time of the next cycle is e_O; if the phase change has occurred, then the expected execution time of the next cycle is e_B. The decision to remap initially has two components. A new mapping is calculated on the basis of the computation's recent behavior. We then suppose that it is possible to estimate the performance of the computation under the new mapping (this issue is discussed in [46]). We can then compare the performance of the new mapping with the performance of the old, and implicitly verify whether the phase change has occurred. The delay associated with calculating and testing a new mapping is D_d. If the testing phase reveals that the phase change has not occurred, then the new mapping is rejected, the probability of a phase change having already occurred is set to zero, and the decision process continues as before. If the new mapping is accepted, then a delay D_r is required to implement it. The decision process then stops, with the expectation that every remaining cycle has a mean execution time of e_R.

We place the remapping decision problem in the context of a Markov decision process [47] by defining the state of the process at the *nth* decision step to be $<p_n,n>$, the probability of the phase change having already occurred coupled with the decision step identifier n. The immediate costs of the possible decisions have been identified as mean per cycle execution costs, and remapping overhead costs. The cost of a decision policy is the expected sum of its decision costs; in our formulation that sum is the expected execution time of the computation (including any remapping overhead). An optimal decision policy is one which minimizes that expected sum. The optimal decision policy is determined by the *optimal cost function* $V(<p_n,n>)$, which expresses the expected future costs of using the optimal decision policy. By the principle of optimality, the optimal cost function is expressed by

$$V(<p_n,n>) = \min \begin{cases} ER(<p_n,n>) \\ EC(<p_n,n>) \end{cases}$$

where $ER(<p_n,n>)$ is the minimized expected future costs given that remapping is chosen from state $<p_n,n>$ and $EC(<p_n,n>)$ is the minimized expected future costs given that remapping is not chosen from state $<p_n,n>$. The optimal decision to make from state $<p_n,n>$ is the decision with the minimum value of the expected future costs function. In [46] it is shown that the optimal decision policy for our problem has a particularly nice form. This result is stated as Theorem 1.

Theorem 1 : For every decision step n, there exists a threshold π_n such that the optimal decision at step n is to choose remapping if and only if $p_n > \pi_n$.

Theorem 1 appeals to our intuition: to remap, we must be sufficiently certain that a phase change has occurred. The magnitude of our required confidence is reflected in the value of the thresholds $\{\pi_n\}$, which are a function of all the model parameters we have identified.

While Theorem 1 gives a nice form for the optimal decision policy, there are severe problems which prohibit its general use. In [46] we show that solving for the $\{\pi_n\}$ is computationally intractable. Furthermore, that solution depends on the quantification of model parameters we may not be certain of. In particular, it may be unreasonable to expect a good estimate of the per cycle gain from remapping. We addressed these concerns by conducting an empirical study which showed that optimal performance can be nearly achieved without explicitly calculating the $\{\pi_n\}$, and without precisely estimating model parameters. This study focused on a simple decision heuristic whose performance is close to that of the optimal policy. We now outline the details of the heuristic.

Like the optimal decision policy, the decision heuristic maintains the probability of change, and reacts to high values of this probability. A threshold ρ is chosen so that ρ is approximately equal to the probability of change which would occur after three successive phase change tests report a change. When the probability of change exceeds ρ, a new mapping is computed. However, this mapping is not

necessarily adopted immediately. The parameters e_B and e_R are estimated to determine the gain from remapping. Then, as a function of these and all other model parameters, a time threshold n_0 is computed as outlined in [46]. n_0 represents a "break-even" point, $\pi_n = 1$ for all $n \geq n_0$, and $\pi_n < 1$ for $n < n_0$. This means that if the computation is close enough to termination, the optimal decision policy will not remap even if the probability of change is 1. Likewise, our heuristic will not adopt a remapping if $m \geq n_0$, m being the current step. Supposing that $m < n_0$, our heuristic calculates pseudo-optimal thresholds $\{\tau_n\}$ for the steps between m and n_0. These thresholds are found by a linear interpolation between $\tau_m = 0.8$ and $\tau_{n_0} = 1.0$. The decision process then proceeds as though the $\{\tau_n\}$ are the optimal decision thresholds $\{\pi_n\}$.

Our study of the heuristic's performance assumed relatively small values of change test error probabilities ($\alpha = 0.05$, $\beta = 0.2$). It used $\phi = 1/E[M]$, which says (approximately) that we expect that it is as likely as not that a phase change occurs during the computation. We used remapping overhead values of $D_d = D_r = 100$; and we used $e_B = 200$. We therefore implicitly assume that the remapping overhead costs are essentially the same as a single post-change cycle under the original mapping. Our cost structure only takes into account computational costs after a phase change, hence we assumed that $e_0 = 0$. The gain from a new mapping is $G = e_B - e_R$. The study varied G, and it varied the length of the computation. For simplicity, we assumed M to be a degenerate, constant random variable.

The empirical study accurately estimated the optimal thresholds $\{\pi_n\}$ using a method described in [46]. It also calculated the performance obtained if a new mapping is never chosen. Comparing the performance of these two extreme policies, we calculated the percentage $\%_n$ improvement possible using the optimal policy. Using a simulation of our model, we measured the percentage of this gain which is achieved by the heuristic, denoted $\%_H$. Table II gives the values of $\%_n$ and $\%_H$ for varying values of G and M.

Table II

$\%_n$ and $\%_H$ for varying values of G and M.

M	G	$\%_n$	$\%_H$	G	$\%_n$	$\%_H$	G	$\%_n$	$\%_H$
10	5	0.0		50	4.7	55.2	100	19.2	75.3
50	5	0.48	54.8	50	11.3	93.4	100	32.4	95.5
100	5	0.5	82.9	50	12.2	95.1	100	33.9	97.1
1000	5	0.94	98.3	50	12.5	99.5	100	34.3	99.5

Obviously, when the gain achievable by remapping is small, there is very little difference between using the optimal policy, and the policy which does nothing at all. If G is larger, the possible performance gains are larger, and most of those gains are captured by our heuristic. It is also clear that the length of the computation affects performance. As the length grows, our heuristic's performance approaches that of the optimal decision policy.

The data given above assumed that the decision heuristic knew precisely what the value of G was. Because this is unrealistic, we re-examined performance under the assumption that the heuristic grossly under or over estimates the value of G. For each pair of G and M, we caused the heuristic to under-estimate G by factors of 10^{-1}, 10^{-2}, and 10^{-3}. We also caused it to over-estimate G by factors of 10, 10^2, and 10^3. Table III lists the resulting $\%_H$ as a function of G, M, and the estimation error factor. This table clearly shows that the heuristic's performance becomes insensitive to estimation error as the length of the computation increases. When the length of the computation is small and the gain G is not estimated correctly, remapping can lead to a degradation in performance. The negative percentages depicted in table III demonstrate this phenomenon. This data suggests the following conclusion, which is quite important if model parameters cannot be estimated:

If remapping costs are the same order of magnitude as a cycle execution time, and if the number of cycles in a computation is modestly large, then most of the gain possible from

Table III

$\%_H$ as a function of G, M, and the estimation error factor.

G	M	10^{-3}	10^{-2}	10^{-1}	10	10^2	10^3
50	10	-7.1	-16.7	-19.4	44.5	48.3	44.1
100	10	5.2	-3.5	0	77.0	77.1	74.3
5	50	35.2	35.1	34.6	-67.1	-97.7	-108.2
50	50	10.0	23.1	33.8	93.9	94.2	92.4
100	50	14.1	27.0	82.0	95.6	96.4	95.1
5	100	11.4	21.7	22.3	49.6	42.9	42.6
50	100	53.1	50.5	86.9	95.2	96.4	96.1
100	100	55.7	53.2	95.7	97.2	97.5	97.6
5	1000	92.7	92.6	95.3	98.4	98.3	98.4
50	1000	94.8	98.0	99.6	99.7	99.7	99.7
100	1000	95.2	99.2	99.6	99.7	99.7	99.7

remapping is achieved if the phase change is detected relatively accurately.

In particular, when the number of cycles in a computation is reasonably large, it is not critically important to be able to estimate what the remapping gain will be. This observation also supports our decision to model only one phase change. Not only would the explicit modeling of multiple phase changes require more difficult analysis, but our data suggests the extra detail is not important. Good performance depends on the relative costs of remapping, the length of the computation, and the accuracy of phase change detection. Since a high cost of remapping can be amortized over a long enough computation, the dominant concern (for long computations) is accurate detection of a phase change.

7. Summary

The tradeoff between the costs and the benefits of remapping a variety of different kinds of problems are examined. In one case, the time-variant behavior of many scientific computations is characterized by gradual changes in each processor's computational load. Coupled with the computation's synchronization needs, performance declines gradually. Good processor utilization requires that the

computational load be balanced between processors, yet a good balance can't be sustained without periodic, possibly expensive remappings. To treat this type of problem, we need to both model the phenomenon of performance degradation, and develop remapping decision policies which effectively determine when the computational load should be remapped onto the parallel machine. We have done so using two different models of gradual load evolution. We have developed and studied an adaptive remapping decision policy SAR which proves to be effective on both models. SAR does not depend on the details of the model structure; rather, it attempts to minimize a statistic which measures the long-term average system degradation (including that due to remapping) as a function of time. For both load evolution models, the performance obtained through the use of SAR is compared to a variety of policies for scheduling remappings.

We have also considered strategies for deciding when to remap problems whose behavior undergoes unpredictable and sudden phase changes. We treat the problem of determining when and if the stochastic behavior of the workload has changed enough to warrant the calculation of a new partition. The problem is modeled as a Markov decision process, the solution to which is intractable. A heuristic is proposed which triggers a remapping when a phase change has been detected and when it is possible to estimate that the duration of the remaining computation duration is sufficiently large to allow performance gains to outweigh remapping overhead. Simulation studies suggest that most of the possible gains in performance that can be obtained through remapping can be captured by the heuristic. The key conclusion of this study is that the timely detection of phase change is the most important issue in achieving good performance. If phases tend to be long-lived, then the accurate estimation of performance gains and performance overheads is not critical. If phases are short-lived, then the heuristic we describe provides good performance if our model parameters can be accurately quantified.

The two types of remapping problems discussed here share common features. Both problems arise due to the fact that a computation is distributed across a parallel computation, and that the

computation's stochastic behavior is not constant. In both problems a global remapping can temporarily restore good system performance, but the decision to remap must weigh the performance gains against the remapping overhead. The differences in these problems' behavior cause their respective treatments to differ; however, these treatments are similar in that they both are adaptive, and they both explicitly consider the appropriate costs and benefits involved in the decision. For both problems, empirical studies of the proposed policies prove their effectiveness.

Acknowledgments

We thank Bob Voigt for his continuing support. We also thank Roger Smith, Paul Reynolds, Dan Reed and Merrell Patrick for helpful discussions.

REFERENCES

[1] M. J. BERGER and A. JAMESON, Automatic adaptive grid refinement for the euler equations, AIAA Journal, 23 (1985), pp. 561-568.

[2] M. J. BERGER and J. OLIGER, Adaptive mesh refinement for hyperbolic partial differential equations, J. Comp. Phys., 53 (1984), pp. 484-512.

[3] W. D. GROPP, Local uniform mesh refinement with moving grids, Yale Technical Report YALEU/DCS/RR-313, April 1984.

[4] M. M. RAI and T. L. ANDERSON, The use of adaptive grid generation method for transonic airfoil flow calculations, AIAA Paper 81-1012, June 1981.

[5] A. HARTEN and J. M. HYMAN, Self-adjusting grid methods for one-dimensional hyperbolic conservation laws, Los Alamos Report LA-9105, 1981.

[6] W. USAB and E. M. MURMAN, Embedded mesh solutions of the euler equation using a multiple-grid method, Proceedings of the AIAA Computational Fluid Dynamics Conference, Danvers, Mass., Paper 83-1946, July 1983.

[7] M. J. BERGER and S. BOKHARI, The partitioning of non-uniform problems, ICASE Report No. 85-55, November 1985.

[8] W. D. GROPP, Local uniform mesh refinement on loosely-coupled parallel processors, Yale Technical Report YALEU/DCS/RR-352, December 1984.

[9] W. D. GROPP, Dynamic grid manipulation for PDEs on hypercube parallel processors, Yale Technical Report YALEU/DCS/RR-458, March 1986.

[10] C. ANDERSON and C. GREENGARD, On vortex methods, SIAM J. Numer. Anal., 22 (1985), pp. 413-440.

[11] A. LEONARD, Vortex methods for flow simulation, J. Comp. Phys., 37 (1980), pp. 289-335.

[12] S. BADEN, Dynamic load balancing of a vortex calculation running on multiprocessors, Computer Science Technical Report, University of California Berkley, 1986.

[13] D. R. WELLS, Multirate linear multistep methods for the solution of systems of ordinary differential equations, Report No. UIUCDCS-R-82-1093, University of Illinois, July 1982.

[14] D. BAI and A. BRANDT, Local mesh refinement multilevel techniques, Dept. of Appl. Math., Weizmann Institute of Science Report, 1984.

[15] S. McCORMICK and J. THOMAS, The fast adaptive composite grid method for elliptic equations, Math. Comp., 46 (1986), pp. 439-456.

[16] A. BRANDT, Multilevel adaptive solutions to boundary value problems, Math. Comp., 31 (1977), pp. 333-390.

[17] R. E. BANK, A multi-level iterative method for nonlinear elliptic equations, in Elliptic Problem Solvers (Martin Schultz, ed.), Academic Press, New York, 1981.

[18] O. C. ZIENKIEWICZ and A. W. CRAIG, Adaptive mesh refinement and a posteriori error estimation for the p-version of the finite element method, in Adaptive Computational Methods for Partial Differential Equations (Ivo Babuska, ed.), SIAM, Philadelphia, 1983.

[19] G. S. FISHMAN, Principles of Discrete Event Simulation, Wiley & Sons, New York, 1978.

[20] K. M. CHANDY and J. MISRA, Distributed simulation: A case study in design and verification of distributed programs, IEEE Trans. on Software Engineering, SE-5, 5 (September 1979), pp. 440-452.

[21] A. I. CONCEPCION, Distributed simulation on multi-processors: Specification, design, and architecture, Ph.D. Dissertation, Wayne State University, January 1985.

[22] D. R. JEFFERSON and H. SOWIZRAL, Fast concurrent simulation using the time warp mechanism, Rand Report to the Air Force, FN-1906-AFFR, December 1982.

[23] J. K. PEACOCK, E. MANNING, and J. W. WONG, Synchronization of distributed simulation using broadcast algorithms, Computer Networks, 4 (1980), pp. 3-10.

[24] P. F. REYNOLDS, Jr., A shared resource algorithm for distributed simulation, Proceedings of the Ninth Annual International Computer Architecture Conference, Austin, Texas (1982), pp. 259-266. Distributed Comp.

[25] D. M. NICOL and P. F. REYNOLDS, Jr., The automated partitioning of simulations for parallel execution, Technical Report No. TR-85-15, Department of Computer Science, University of Virginia, August 1985.

[26] D. L. EAGER, E. D. LAZOWSKA, and J. ZAHORJAN, Adaptive load sharing in homogeneous distributed systems, IEEE Trans. on Software Eng., SE-12 (May 1986), pp. 662-675.

[27] G. J. FOSCHINI, On Heavy Traffic Diffusion Analysis and Dynamic Routing in Packet Switched Networks, Computer Performance (K. M. Chandy and M. Reiser, eds.), New York: North-Holland.

[28] L. M. NI, C. XU, and T. B. GENDREAU, A distributed drafting algorithm for load balancing, IEEE Trans. on Software Engineering (October 1985), pp. 1153-1161.

[29] J. A. STANKOVIC, An application of Bayesian decision theory to decentralized control of job scheduling, IEEE Trans. on Computers, C-34, 2 (February 1985), pp. 117-130.

[30] J. A. STANKOVIC, K. RAMAMRITHAM, and S. CHENG, Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems, IEEE Trans. on Computers, C-34, 12 (December 1985), pp. 1130-1143.

[31] D. TOWSLEY, Queueing network models with state-dependent routing, Journal of the ACM, 27, 2 (April 1980), pp. 323-337.

[32] A. N. TANTAWI and D. TOWSLEY, Optimal static load balancing, Journal of the ACM, 32, 2 (April 1985), pp. 445-465.

[33] W. W. CHU, L. J. HOLLOWAY, M. LAN, and K. EFE, Task allocation in distributed data processing, Computer, 13, 11 (November 1980), pp. 57-69.

[34] A. DUTTA, G. KOEHLER and A. WHINSTON, On optimal allocation in a distributed processing environment, Management Science, 28, 8 (August 1982), pp. 839-853.

[35] D. GUSFIELD, Parametric combinatorial computing and a problem of program module distribution, Journal of the ACM, 30, 3 (July 1983), pp. 551-563.

[36] H. S. STONE, Critical load factors in distributed computer systems, IEEE Trans. on Software Engineering, SE-4, 3 (May 1978), pp. 254-258.

[37] J. A. BANNISTER and K. S. TRIVEDI, Task allocation in fault-tolerant distributed systems, Acta Informatica, 20 (1983), pp. 261-281.

[38] S. BOKHARI, Partitioning problems in parallel, pipelined, and distributed computing, ICASE Report No. 85-54, November 1985.

[39] M. A. IQBAL, J. H. SALTZ, and S. H. BOKHARI, Performance tradeoffs in static and dynamic load balancing strategies, to appear in the Proceedings of the 1986 International Conference on Parallel Processing.

[40] P. J. DENNING, Working sets past and present, IEEE Trans. on Software Engineering, SE-6, 1 (January 1980), pp. 64-84.

[41] J. R. SPIRN, Program Behavior: Models and Measurement, Elsevier North-Holland Inc., New York, 1977.

[42] D. M. NICOL and J. H. SALTZ, Dynamic remapping of parallel computations with varying resource demands, ICASE Report 86-45, June 1986.

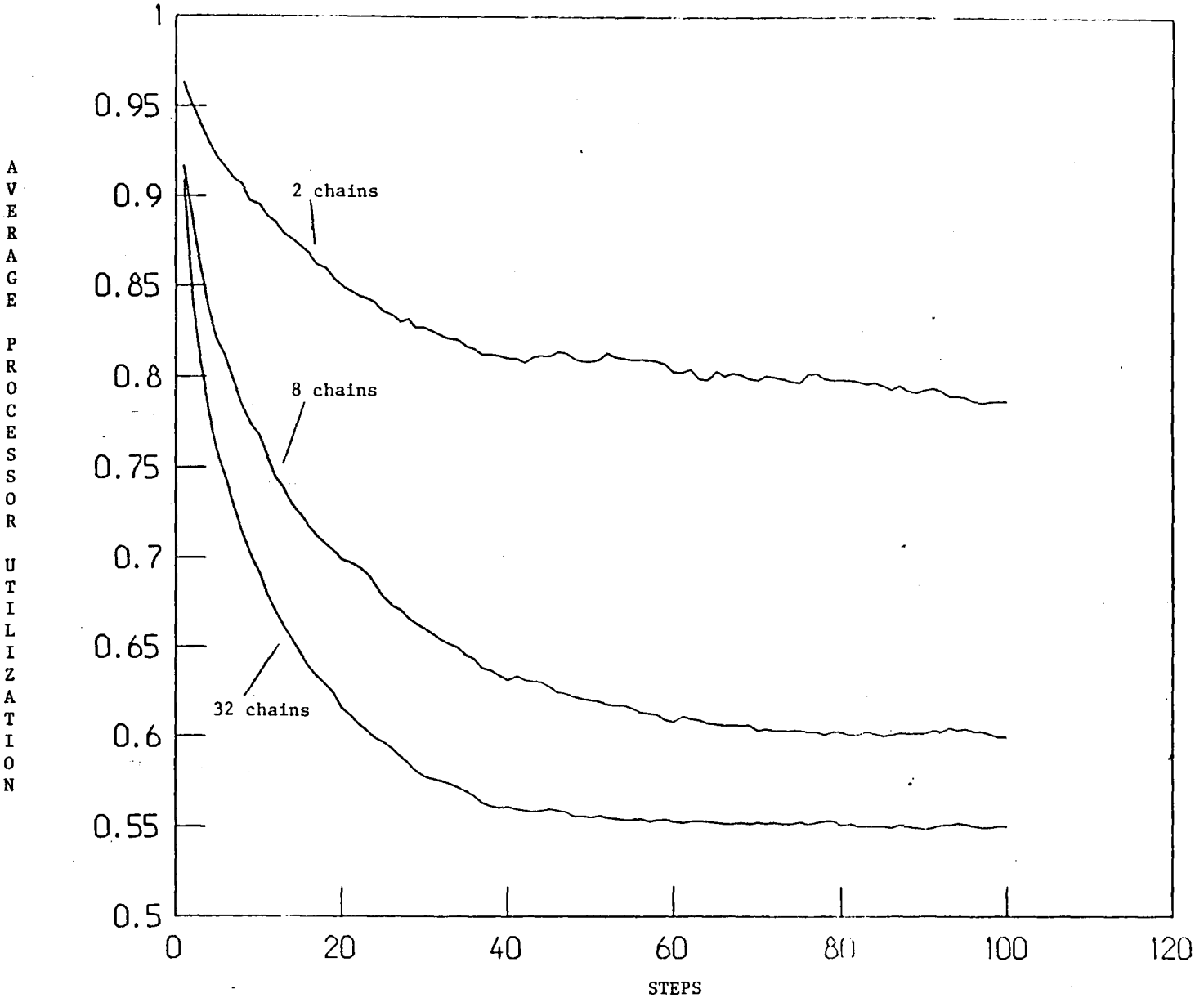
[43] D. SACCA and G. WIEDERHOLD, Database partitioning in a cluster of processors, ACM Transactions on Database Systems, 10 (1983), pp. 29-56.

[44] S. NAVATHE, S. CERI, G. WIEDERHOLD, and J. DOU, Vertical partitioning algorithms for database design, ACM Database, 9 (1984), pp. 680-710.

[45] L. W. DOWDY and D. V. FOSTER, Comparative models of the file assignment problem, ACM Surveys, 14 (1982), pp. 287-313.

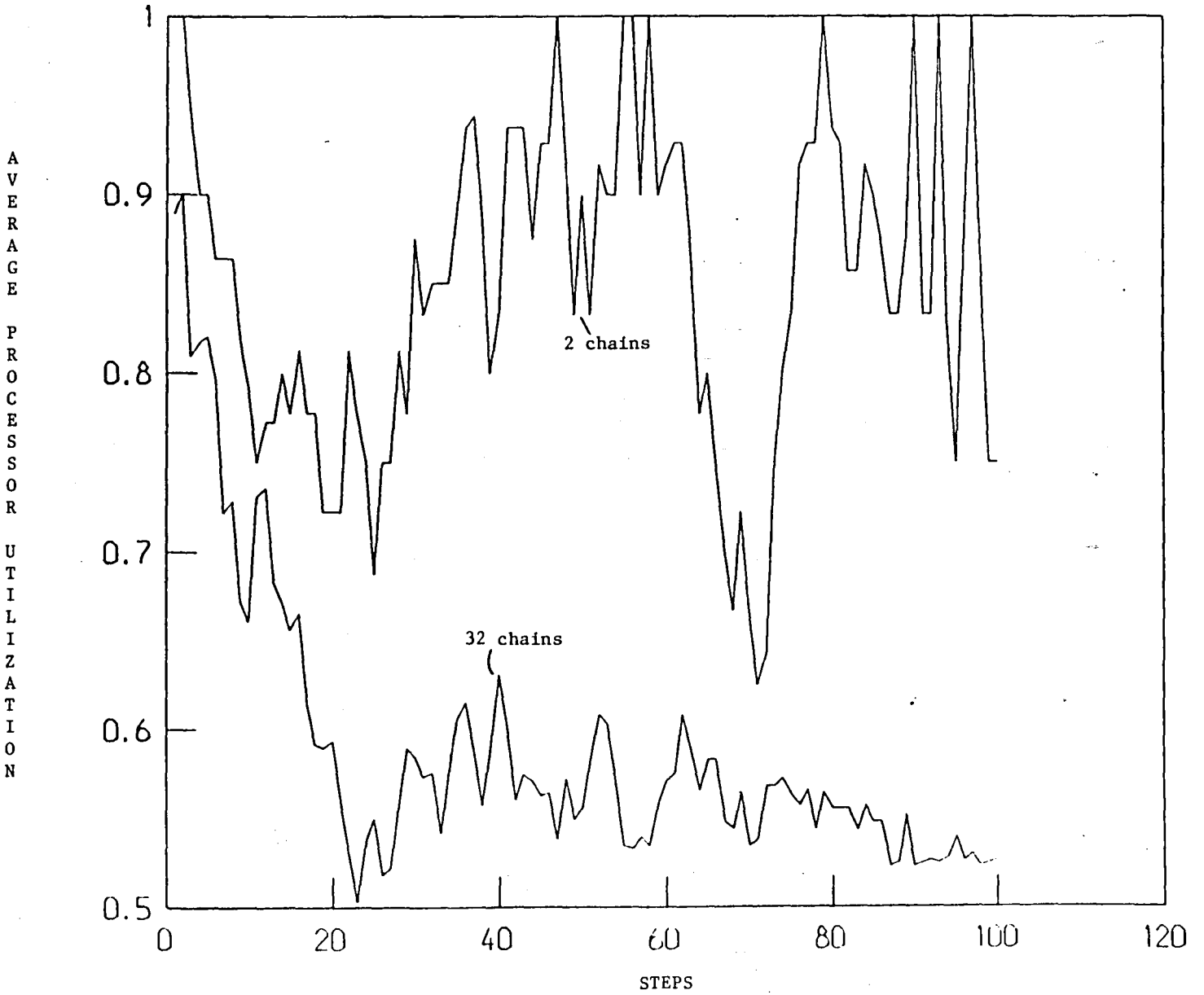
[46] D. M. NICOL and P. F. REYNOLDS, Jr., An optimal repartitioning decision policy, ICASE Report 86-7, February 1986.

[47] S. ROSS, Applied Probability Models with Optimization Applications, Holden and Day, San Francisco, 1971.



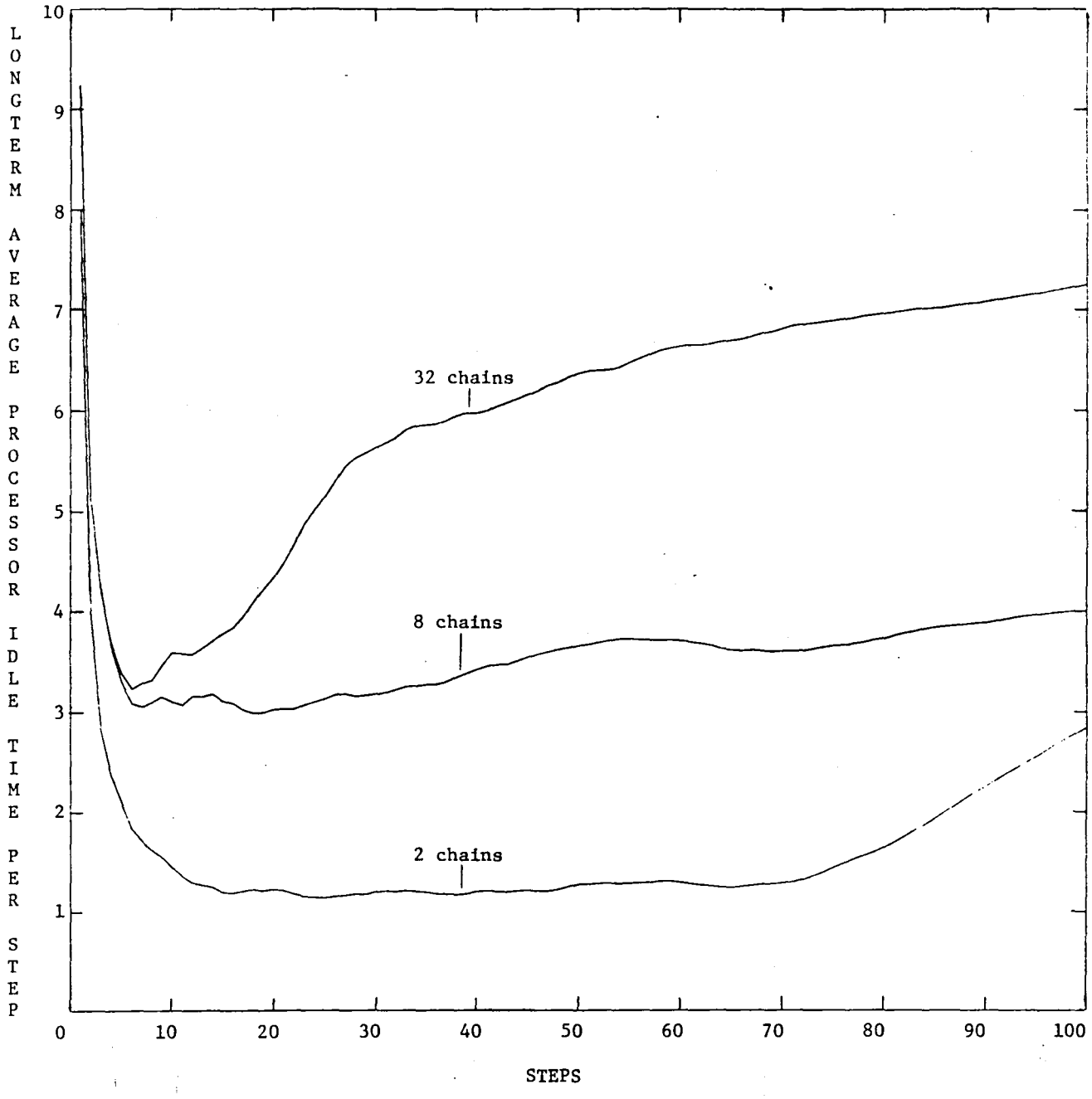
MUM Model: Expected Average Processor Utilization as a Function of Step Estimated from 500 Sample Paths. Each chain has 19 states, p = 0.5.

Figure 1a



MUM Model: Average Processor Utilization as a Function of Step-Single Sample Path.
Each chain has 19 states, p = 0.5.

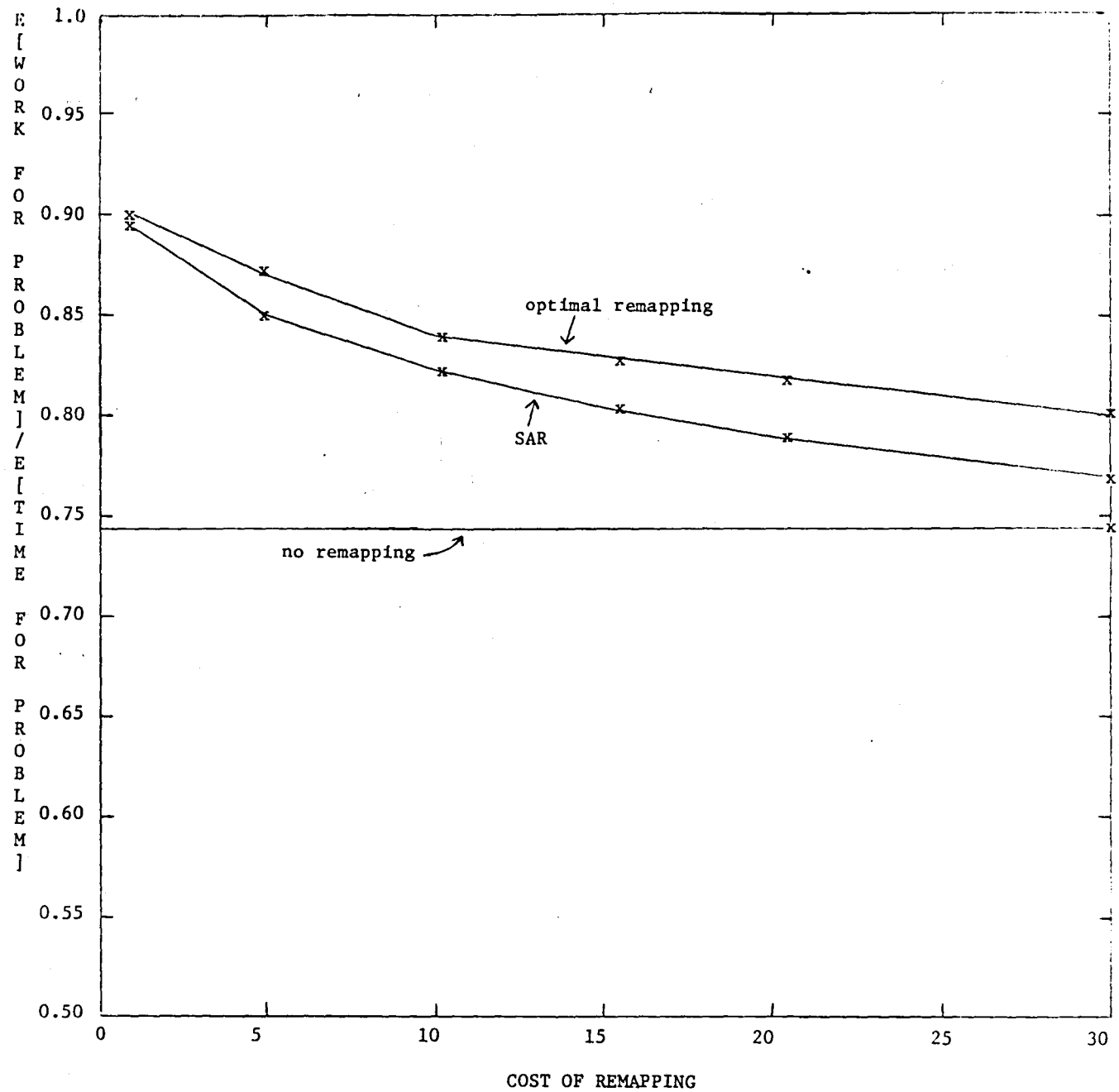
Figure 1b



MUM Model: Longterm Average Processor Idle Time per Step w(n) from Single Sample Path.

Each chain has 19 states, p = 0.5, load balancing cost = 8.

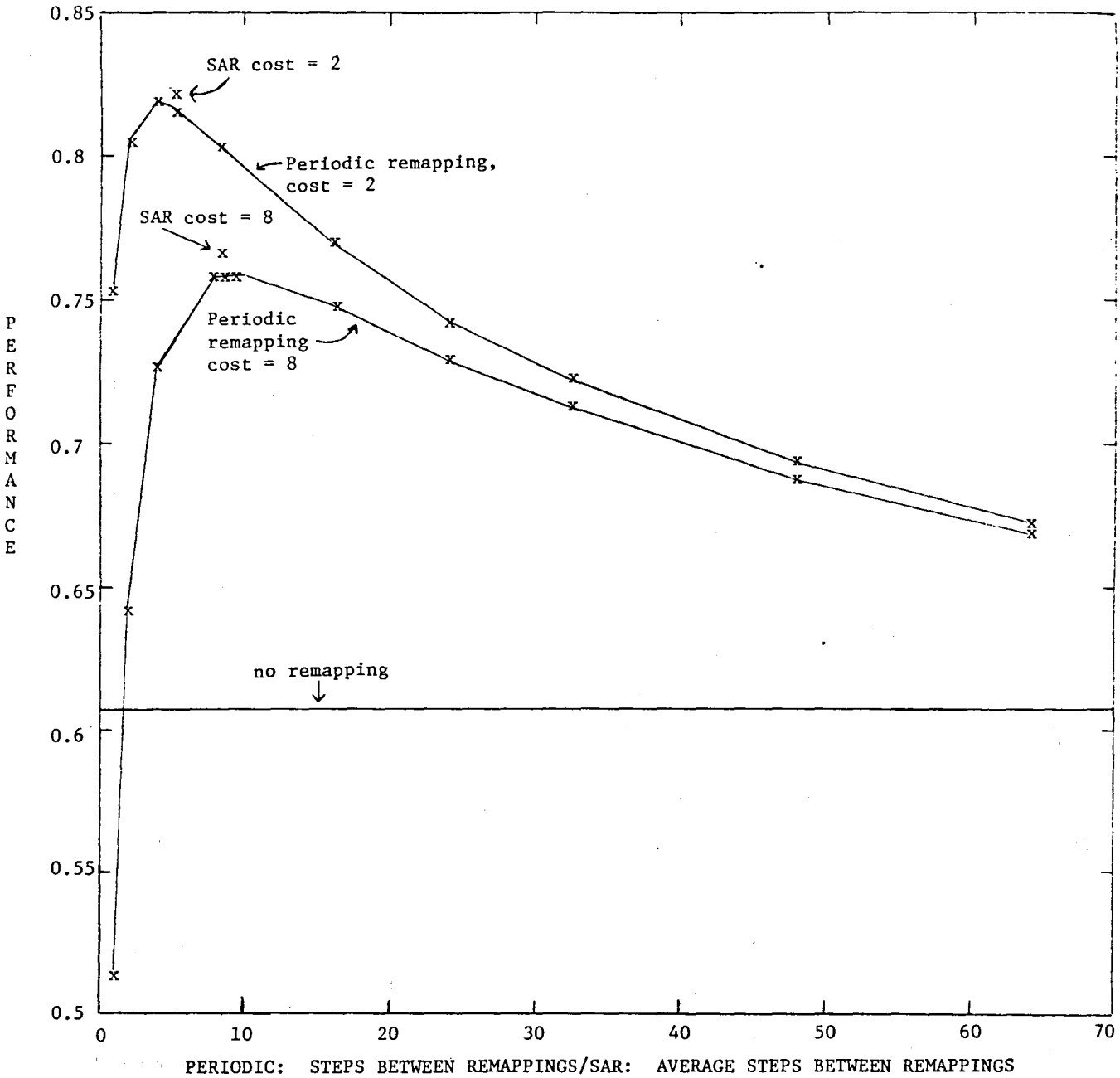
Figure 2



MUM Model: Performance of Optimal Remapping Decision Policy Compared with Performance of SAR.

Three chains, 100 steps, each chain has 19 states, p = 0.5, SAR performance calculated from 500 sample paths.

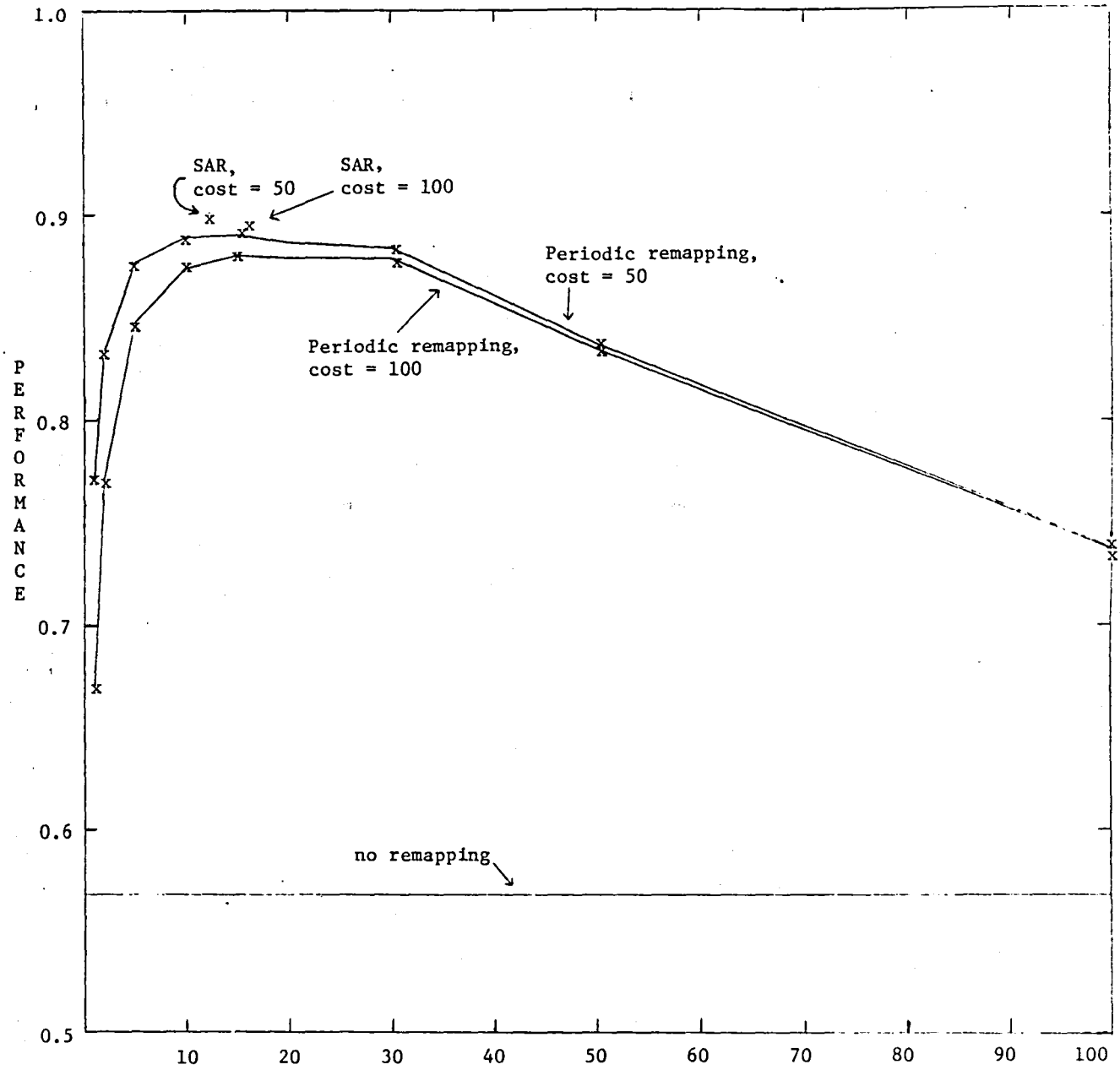
Figure 3



MUM Model: Performance of SAR Compared with Performance of Periodic Remapping.

Eight chains, 400 steps, each chain has 19 states, p = 0.5, each data point calculated from 200 sample paths.

Figure 4



LD Model: Performance of SAR Compared with Performance of Periodic Remapping

64 by 64 activity array initialized with one work unit per activity point. Work unit transition probabilities: up - 0.1, right - 0.1, down - 0.05, left - 0.05. Each data point calculated from 50 sample points.

Figure 5

Standard Bibliographic Page

1. Report No. NASA CR-178129 ICASE Report No. 86-46	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle STATISTICAL METHODOLOGIES FOR THE CONTROL OF DYNAMIC REMAPPING		5. Report Date July 1986 6. Performing Organization Code
7. Author(s) Joel H. Saltz, David M. Nicol		8. Performing Organization Report No. 86-46
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 11. Contract or Grant No. NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report 14. Sponsoring Agency Code 505-31-83-01

15. Supplementary Notes

Langley Technical Monitor:
J. C. South

Submitted to Proceeding ARO

Final Report

16. Abstract

Following an intial mapping of a problem onto a multiprocessor machine or computer network, system performance often deteriorates with time. In order to maintain high performance, it may be necessary to remap the problem. The decision to remap must take into account measurements of performance deterioration, the cost of remapping, and the estimated benefits achieved by remapping. We examine the tradeoff between the costs and the benefits of remapping two qualitatively different kinds of problems. One problem assumes that performance deteriorates gradually, the other assumes that performance deteriorates suddenly. We consider a variety of policies for governing when to remap. In order to evaluate these policies, statistical models of problem behaviors are developed. Simulation results are presented which compare simple policies with computationally expensive optimal decision policies; these results demonstrate that for each problem type, the proposed simple policies are effective and robust.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
multiprocessor load balancing, dynamic load balancing, dynamic remapping, message passing, distributed computation, adaptive regridding, simulation	61 - Computer Programming and Software Unclassified - unlimited

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	37	A03

For sale by the National Technical Information Service, Springfield, Virginia 22161

End of Document